RESEARCH PLAN PROPOSAL

DESIGN AND ANALYSIS OF MODEL TO STUDY THE IMPACT OF REFACTORED CODE ON SOFTWARE PRODUCT LINE MAINTAINABILITY

For registration to the degree of
Doctor of Philosophy

IN THE FACULTY OF SCIENCE

THE IIS UNIVERSITY, JAIPUR

Submitted by

Uma Devi
Enroll. No. – ICG/2013/16507

Under the Supervision of

Supervisor

Dr. Nishtha Kesswani
Assistant Professor,
Central University of Rajasthan,
Rajasthan

Co-Supervisor

Dr. Amita Sharma
Assistant Professor,
THE IIS UNIVERSITY,
Jaipur (Raj).

Department Of Computer Science

May 2014

TABLE OF CONTENTS

1. Topic

2. Introduction

    2.1 Classical Software Engineering

        2.1.1 Problems in Classical Software Engineering

        2.1.2 Evolution of New Methodologies and Methods

    2.2 Software Product Lines

    2.3 Feature Oriented Programming

    2.4 Delta Oriented Programming

    2.5 Code Clones

        2.5.1 Code Clones in Object Oriented Programming and Feature Oriented Programming

        2.5.2 Classification of Code Clones

        2.5.3 Effects of Code Clones on Quality of Software Product Lines

    2.6 Quality

        2.6.1 Metrics

        2.6.2 Quality Models

    2.7 Refactoring as a Remedial Measure for Cloning

3. Literature Review

4. Motivation/ Justification and Relevance

5. Objectives

6. Plan of Work and Methodology

7. Expected Outcomes of the Proposed Research Work

8. Place of Work and Facilities Available

9. References, Bibliography, Webliography

1. TOPIC

Design and Analysis of Model to study the impact of Refactored code on Software Product Line Maintainability

2. INTRODUCTION

2.1 Classical Software Engineering

Classical software engineering approach has a lot of models that deal with different types of projects such as waterfall, spiral, iterative but all of them and other lack flexibility. There are a number of phases common to every development, regardless of methodology, starting with requirements capture and ending with maintenance. With the traditional approach, it will be expected to move forward slowly and consecutively from one phase to the other. In classical software engineering, software construction consists of two main activities performed sequentially: **development and maintenance**. Beginning from scratch, the software product is developed and then installed on the client's computer. Any change to the software after installation, whether to fix a residual fault or extend the functionality, constitutes maintenance. Thus, the way that software was developed classically can be described as the development-and-then-maintenance model [1].

2.1.1 Problems in Classical Software Engineering

However, classical software engineering model has proved to be impractical for two reasons. First, software engineers usually have to cope with changing requirements and evolving technologies, the so - called “moving target problem”. In other words, developers frequently have to perform software maintenance before the product is installed. In theory, such maintenance might have to be performed just one day after the client’s original requirements were given to the software engineering team, in contrast to the classical model of development-and-then-maintenance. Software reuse is the second reason why development -and-then-maintenance is no longer an accurate model for software construction. The limitation of classical software engineering paradigm are rooted in a focus on sequential thinking which focuses on the activities presumed to be necessary to produce a result. The introduction of software engineering as an engineering discipline has led to formal methodologies for developing software applications and has encouraged logical thinking [2].

2.1.2 Evolution of New Methodologies and Methods

A number of different methodologies, methods, and techniques for developing software were reviewed and characterized according to a number of different attributes (e.g. history, process, artifacts, iteration length, project management, scope within SDLC and applicability in different contexts) [3].

Methodologies: The basic common methodologies for software development lies under the categories of :

- **Unified Process Methodology :** Developed as a process to accompany the Unified Modeling Language (UML), a set of mostly graphic models covering all phases of software development.

- **Crystal Family Methodology:** The methods within the family are chosen based on several factors such as team size, system criticality and project priorities.

Methods: The different methods used in software engineering are:

- **Extreme Programming Method:** It is a programming-focused software development method that intends to improve software quality and responsiveness to changing customer requirements through simplest coding solutions.
- **Scrum Method:** It is a project management method that concentrates on how team members should function in order to develop software flexibly in a constantly changing environment.
- **Feature Driven Development Method:** It is an iterative and incremental software development method that is model centric, business requirements-driven and emphasizes quality through the processes.
- **Dynamic System Development Method:** The goal of which is to deliver projects on time and on budget while adapting to changing requirements.
- **Adaptive Software Development Method:** It involves iterative and incremental development with constant prototyping and encourages continuous adaptation of the processes.

Software Product Line development refers to software engineering methods, tools and techniques for creating a collection of similar software systems from a shared set of software assets using a common means of production.

2.2 Software Product Lines

The software architecture is the first artifact to place requirements in a solution space. The software architecture is defined by Bass (1999) as follows, "*The software architecture of a program or computing system is the structure or structures of the system, which comprise software components, the externally visible properties of those components, and the relationships among them*"[4]. Over the last few years a new approach to software reuse has become very popular in industry and academia. This approach is known as **Software Product Line Development** and it supports large-grained intra-organization software reuse. The basic idea of this approach is to use business domain knowledge to separate the common parts of a

family of products from the differences between the products. The commonalties are used to create a product platform that can be used as a common baseline for all products within a product family [5]. Software product line development involves software reuse. In product line development reuse is planned, enabled and enforced. The reuse repository of a software product line is known as the core assets of the product line. The core assets include all the artifacts that are the most costly to develop; domain models, requirements, architecture, components, test cases, and performance models, etc. The activities and aspects peculiar to product lines are requirement elicitation, requirement analysis, requirement specification, requirement validation, requirement management etc.

2.3 Feature Oriented Programming

A ***feature*** is a unit of functionality of a software system that satisfies a requirement, represents a design decision, and provides a potential configuration option. ***Feature-oriented software development* (*FOSD*)** is a paradigm for the construction, customization, and synthesis of large-scale software systems. The concept of a feature is at the heart of FOSD. FOSD aims essentially at three properties: **structure, reuse, and variation** [6]. The basic idea of FOSD is to modularize software into feature modules which represent features. To create an application, feature modules are composed. In SPLs features that are common to multiple products are identified and managed. The dependencies between the features are also captured and constraints put on the combination of features. The features are then used in the implementation of different concrete products. This introduces flexibility to compose features in different combinations, e.g., omit certain features or implement alternative features.

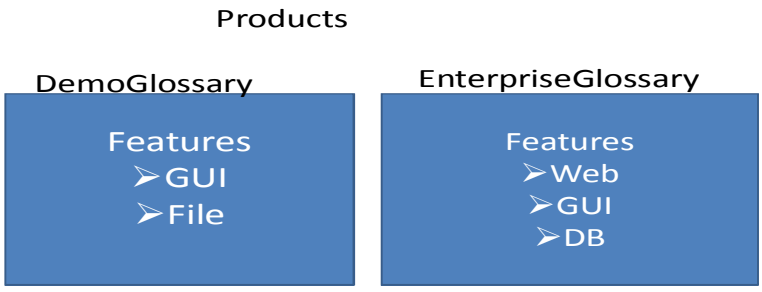


Figure 1(a): Overview of the glossary product line

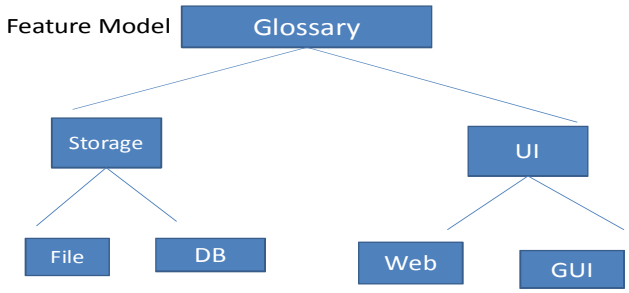


Figure 1(b): Overview of the glossary product line

Figure 1(a) , 1(b) depicts the feature model of the product line, the features of the products and an excerpt of the development artifacts that realize the product line

* S. Apel and C. Kastner. "**An overview of feature-oriented software development**". *Journal of Object Technology*. Volume 8, Issue 5. pp. 49-84, 2009.

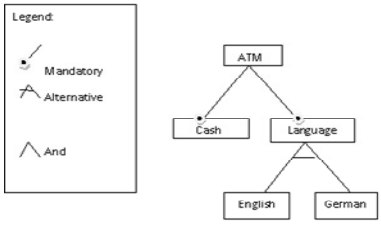


Figure 2: Feature Diagram of an ATM SPL

2.4 Delta Oriented Programming

Delta Oriented Programming is a novel programming language approach. A product line is represented by a **core module** and a **set of delta modules**. In DOP, the implementation of a SPL is divided into a core module and a set of delta modules [7]. The goal of DOP is to relax the restrictions of FOP and to provide an expressive and flexible programming language for SPL. The implementation of an SPL can be categorized into a core module and a set of delta modules. The comparison between Feature Oriented and Delta Oriented shows as to how they differ in their **expressiveness, the treatment of domain features, solutions for the optional features problem** etc. It is also seen that to implement SPLs in the object-oriented paradigm, they can be classified as: **annotative approaches**, such as conditional compilation, which mark the source

Figure 2 Example of a Feature Diagram where there are primarily Compound Features (Languages) and Primitive Features (English)

*S. Schulze, O. Richers, and I. Schaefer. "**Refactoring delta-oriented software product lines**". In *AOSD*. pp. 73-84. ACM, 2013

code of the whole SPL with respect to product features on a syntactic level. And **compositional approaches**, such as DELTAJAVA, associate code fragments to product features that are assembled to implement a particular feature configuration [8]. As the source code develops, the probability of duplication of code becomes more prominent and can be found in many segment of the program. This is known as Code Clones or Code Smells. It is one of the crucial problems faced during evolution of Software Product Lines which unfavorably affects the software quality factors especially Maintainability which further leads to higher development costs and decreased

performance. Thus it becomes essential to address the code clones from the beginning of software product line development.

2.5 Code Clones

Code cloning, that is, the replication of code fragments in source code, is known to be a serious and common problem in object oriented programming (OOP). It is widely accepted that code clones have a negative effect on the software system, in terms of decrease of maintainability and the introduction of errors [9]. Code Clones or Code smells can aid the identification of SPL refactorings and it improves the evolvability and maintainability of delta-oriented SPLs.

2.5.1 Code Clones in Object Oriented Programming and Feature Oriented Programming

Programming paradigms such as feature-oriented programming (FOP) aim at removing these limitations which have known to cause the occurrence of code clones in object oriented programming at the first place. After conducting studies it was found that there is a considerable amount of clones in feature-oriented software product lines and that a large fraction of these clones is FOP-related (i.e., caused by limitations of feature-oriented mechanisms).

2.5.2 Classification of Code Clones

a) Code fragments that are (almost) identical are called **Type-I clones**. Only minor differences regarding formatting such as comments or whitespaces are allowed.

b) A common pattern of cloning is Copy & Paste-and-Modification, which leads to **Type-II clones**. These clones diverge more than Type-I clones so that even differences in names of identifiers, literals, types, layout, or comments are included in this type of clones.

c) **Type-III clones** additionally allow changing, adding, or deleting statements. Since deleting a statement from one code fragment can be also interpreted as adding to the corresponding (cloned) statement , both terms (deleting and adding statements) are

treated synonymously. Type-III clones are also referred to as *gapped clones*, where the missing statements are called gaps.

d) **Type-IV clones** can be syntactically different: The cloning relation for this clone type is based on the semantic similarity between two or more code fragments and thus they are also called *semantic clones* [10].

But it is possible to remove code clones with the application of Refactoring Technique. Though, it has not been evidently proved that the problem of code clones is completely eliminated in delta oriented approach, but there is a probability of code clone occurrence in delta oriented programming.

2.5.3 Effect of Code Clones on Quality of SPLs

The causes of code clones in FOP have not yet been quantified and there is still work going on to identify code clones in Delta Oriented Methodology too, but it is evident that code clones effect the quality of SPLs in general which in turn greatly effects the quality factors of a software product and hence results in adverse effects like deterioration in performance, introduction of bugs and errors, decrease in performance and high cost due to poor maintainability.

2.6 Quality

The quality model distinguishes three different views on software product quality:

a) **Internal quality :** concerns the properties of the system that can be measured without executing it
b) **External quality :** concerns the properties of the system that can be observed during its execution
c) **Quality in use :** concerns the properties experienced by its (various types of users) during its operation and maintenance.

The different parameters for assessing software product qualities [11] especially Maintainability are :

a) **Analysability:** how easy or difficult is it to diagnose the system for deficiencies or to identify the parts that need to be modified?
b) **Changeability:** how easy or difficult is it to make adaptations to the system?
c) **Stability:** how easy or difficult is it to keep the system in a consistent state during modification?
d) **Testability:** how easy or difficult is it to test the system after modification?
e) **Maintainability conformance:** how easy or difficult is it for the system to comply with standards or conventions regarding maintainability?

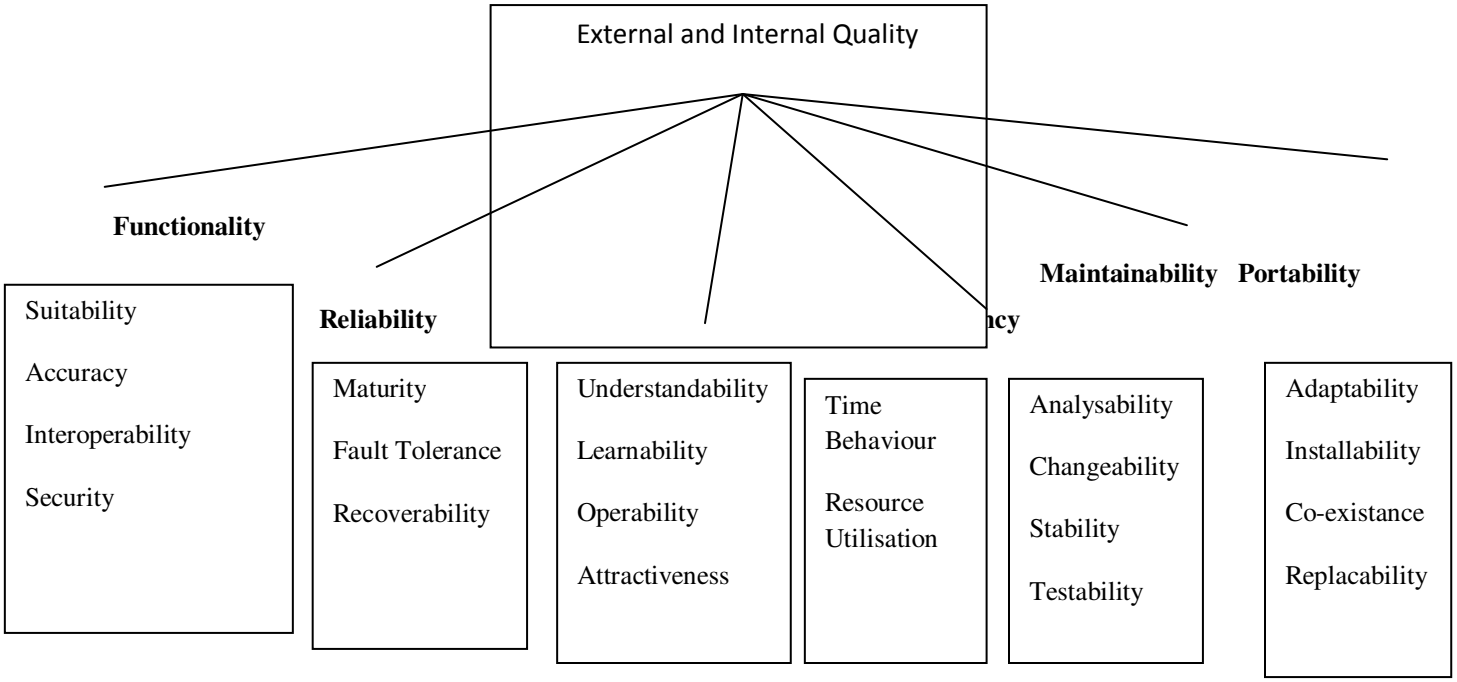


Diagram 3: Breakdown of the notions of external and internal software product quality into 6 main characteristics and 27 sub characteristics.

* I. Heitlager, T. Kuipers, and J. Visser. "**A practical model for measuring maintainability**" In *6th Int. Conference on the Quality of Information and Communications Technology (QUATIC 2007).* pp. 30-39. IEEE Computer Society, 2007

2.6.1 Metrics

When software engineering aspects are to be measured in terms of quality, metrics are considered for measuring complexity and quality, estimating cost and project effort etc. There have been metrics proposed for Object Oriented paradigm previously and researchers have now also validated these metrics both theoretically and empirically [12], such as: Weighted methods

per class (WMC), depth of Inheritance Tree(DIT), number of children (NOC), coupling between objects (CBO), response for a class(RFC) and lack of cohesion in methods (LCOM). The feature based measurements would permit user to select feature based on the expected contribution the feature makes to the final quality of SPL. It assesses the difficulty in measuring Maintainability Index by directly measuring the feature. Algebraic Hierarchical Equations for Application Design (AHEAD) is a model of FOP where each feature implementation encapsulates the set of files realizing its functionality. MI is also provided where future product maintainability is to be assessed [13].

Maintainability Metrics for UML design includes an association between a class and an interface, implementation of an interface by a class, dependency between a class and interface and aggregation of an interface in a class [14].

Metrics for source code is LOC which counts the number of codes excluding whitespaces and number of comments [15].

Metric CLON (CLONICITY), is a ratio of LOC in all detected clones to the total LOC.

Software metrics provide a quantitative basis for planning and predicting software development processes. Therefore the quality of software can be controlled and improved easily. Quality models have become a well-accepted means to describe and manage software quality. A variety of quality models have been developed. Many of these models are used, for example to aid the specification of quality requirements, to assess existing systems or to predict the defect density of a system in the field.

2.6.2 Quality Models

There are five methods of quantifying software maintainability from software metrics [16]. They are:

- ***Hierarchical multidimensional assessment models*** view software maintainability as a hierarchical structure of the source code's attributes.

- ***Polynomial regression models*** use regression analysis as a tool to explore the relationship between software maintainability and software metrics
- ***An aggregate complexity measure*** gauges software maintainability as a function of entropy
- ***Principal components analysis*** is a statistical technique to reduce collinearity between commonly used complexity metrics in order to identify and reduce the number of components used to construct regression model
- ***Factor analysis*** is another statistical technique wherein metrics are orthogonalized into unobservable underlying factors, which are then used to model system maintainability

The **Goal Question Metric (GQM)** paradigm is a mechanism for defining and evaluating a set of operational goals, using measurements. It represents a systematic approach for tailoring and integrating goals with models of software processes, products and quality perspectives of interests, based upon the specific needs of the projects and the organization [17]. Since quality, especially Maintainability of a software product line is greatly affected by code clones, it is necessary to eliminate them as the product line develops.

2.7 Refactoring as a Remedial Measures for Cloning

Refactoring is one popular and promising technique to eliminate the problem of Code Clones. Refactoring is the process of changing a software system in such a way that it does not alter the external behavior of the code yet improves its internal structure. It is a disciplined way to clean up code that minimizes the chances of introducing bugs. Examples of refactoring includes Extract Method and Pull Up Field. As such, a refactoring is usually a small change to the software, although one refactoring can involve others. For example, Extract Class usually involves Move Method and Move Field. And by applying appropriate methods code clones can be improved, thereby, improving the efficiency, performance, reuse and maintainability of the software programs. For FOP, features are considered as the source and targets of refactoring rather than classes. Benefits of Refactoring includes [18]:

- It improves the design of software
- It makes software easier to understand
- Helps find bugs thus helps in faster programming

3. LITERATURE REVIEW

D. Simon et. al [19], in their paper, **Evolutionary Introduction of Software Product Lines** have emphasized the significance of Software Product Line for managing the development of software in the industry its benefits over traditional approach have changed the mindset of software companies to adopt software product line approach for their existing products. Here they are proposing a lightweight iterative process supporting the incremental introduction of product line concepts for existing software products by analyzing the legacy code and the products, using feature analysis. However, their approach avoids the more costly architectural reconstruction of legacy system.

R. Mitschke et.al [4], in **Supporting the evolution of Software Product Lines**, states that the aim of Software Product Lines is reusability and evolution of features common to multiple products. For this reason, traceability is an important factor. To achieve this, authors have proposed that each version of an artifact be associated with a specific version of a feature and thus feature dependencies are also explicitly managed. However, an open challenge when developing SPL is to enable controlled evolution of Product Lines. Here, the author proposes feature models combining versioning models in the context of Software Product Line evolution. Traceability between features, products and artifacts is also suggested to ensure consistency and maintainability.

J. Bosh et. al [20] in **Evolution in Software Product Lines**, states that software product line is an approach which favors increased software reuse and reduced development cost .In this paradigm the architecture and a set of reusable components are shared among a family of products. The author arrives at a conclusion that evolution in software product lines is more complex as compared to traditional software development. The reason being new and (at times) contrasting requirements keep evolving from the existing products in the product line and also from new products that are to be integrated. The increase in requirements leads to addition of features and this increases the complexity of the product line.

L. Chen et. al [21] in **Variability Management in Software Product Lines : A Systematic Review**, have identified and explained many issues related to variability and variability modeling in reference to software product line. The authors identified many factors which lead to increase in variability viz. mechanism selection, tool support, implicit dependencies, phase selection, deficiency of feature modeling, unavailability of uniform representation of variability throughout the entire lifecycle. Authors concluded that variability management requires attention and suitable measures be developed for the efficient management of the complexity.

Maben et. al [22] in **Deficiencies in Feature Models**, stated that feature modeling is one approach to deal with complexity. But expressing several requirements in features makes feature models very complex. Focusing the importance on Feature models he clarified that these models are not only used for domain modeling, but also for product derivation in product line development. Thus concluded that complexity should be first measured in feature model and then in the architecture of the product line.

Gacek. C et. al [23] in **Implementing Product Line Variabilities**, states that since software product lines have numerous members, it becomes crucial that they support the description of the product line as a whole as well as its instantiation for the derivation of individual products. He thus concluded that scalability issues as well as supporting and combining various techniques should be addressed in proper and efficient manner.

S.Apel et. al [5] in **An overview of feature-oriented software development**, here discusses that Feature Oriented Software Development is not a single development method or technique but a conglomeration of different ideas, tools, methods and techniques. It is a product characteristic that is used in distinguishing programs with a family of related programs. The authors together have put forward an overview of FOSD. They have also been able to summarize different works in FOSD and establishing relations between different approaches to FOSD. **T. Thum** et.al [24] in **Abstract Features in Feature Modeling**, presents the concept of Abstract Features which are used to structure a feature model that, however, do not have any impact on implementation level. They are also relevant in feature modeling but in current notation they are assumed only implicitly. A conclusion is derived where a mechanism is presented to transform a propositional formula specifying all valid configurations into a propositional formula specifying the set of distinct program variants.

I. Schaefer et.al [25], in **Delta Modeling for Software Architectures**, discussed the idea of delta modeling of software architectures as a uniform modeling formalism for architectural variability in space and time. Refactoring techniques have also been used here which helps in maintaining and improving the quality of variability model which in turn avoids degeneration of product line model under system evolution. Authors have proposed an Architecture Description Language (ADL), Δ-MontiArc, which allows architectural variability in space and time. Behavioral variability within the architectural descriptions can be realized by using deltas on state machines or java source code.

O. Richers [26], in his thesis, **Transformation and Evolution of DeltaJ Software Product Lines**, has presented that a delta module may implement a single feature, several features as well as fraction of features. A DeltaJ program is controlled by application condition which is a propositional formula over the available features of a software product line, which decides if a certain delta module participates in the generation of a specific product. Author has concluded that Automated program transformations can be used to support the development and evolution of Programs, especially DeltaJ, using Refactorings.

S.Chiba et.al [27] in their paper, **Feature Oriented Programming with Family Polymorphism**, have proposed a new FOP language named FeatureGluonJ, which supports family polymorphism with revisers. The code clones among alternative features are separated into another feature and can be shared among the alternative features by extending that feature. Furthermore, clones in derivatives can be removed as well. It is concluded in this paper that although an approach called software product line (SPL) enables to implement products efficiently by reusing most of their code, SPLs implemented by FOP contain a lot of code clones. Code clones are often caused by alternative features and clones in derivatives are also found among alternative features. Family polymorphism is a traditional approach to reuse structure consisting of classes by extending multiple classes at once.

C.Kastner et.al [8] in **Code Clones in Feature Oriented Software Product Lines**, conducts an empirical study on some feature oriented software product lines with respect to code cloning and arrived at a conclusion that even though Feature Oriented Programming paradigm aims at alleviating the problem of code clone in object-oriented mechanism, due to some limitations, there are still considerable amount of clones in feature oriented software product lines and

majority of it are FOP related. Though the FOP related clones could be removed for an exemplary product line by applying refactoring technique, the causes which cause code clones could not be quantified. **S. Schulze** [9], in his thesis, **Analysis and Removal of Code Clones**, has mainly focused on code clone analysis and removal and the reasons why code clones occur. A refactoring technique for variant preserving nature for compositional software product lines has also been proposed as a means of removing code clones. It has been concluded that there exists the problem of code clones in SPLs in general, but till now it hasn't been able to judge the harmfulness of code clones in software product lines. **S. Thummalapenta** et.al [28] in their paper proposes an automatic approach to categorize the evolution of code clone fragments and inquires the reason as to what extent clones are consistently propagated or they evolve independently.

I. Heitlager et.al [10] in **A Practical Model for Measuring Maintainability**, puts forward the idea that the amount of effort needed to maintain a software system is directly related to the technical quality of the source code of that system. The ISO 9126 model for software product quality recognizes maintainability as one of the 6 main characteristics of software product quality, with adaptability, changeability, stability, and testability as sub characteristics of maintainability. To conclude, the ISO 9126 standard is a good frame of reference for software product quality but it fails to provide a practical method for quality assessment. Also the vast literature shows no concrete ways of measuring software for assessing its quality.

S. Senkaya et. al [29] in **Predicting Quality Attributes of Software Product Lines using Software and Network Measures and Sampling**, proposes a systematic approach which targets at efficient and scalable prediction of software quality attributes of products based on software and network measures and receiver operating characteristic analysis. These predictors are used to make the process of finding 'acceptable' products more efficient using sampling algorithms.

G. Aldekoa et.al [12] in **Experience Measuring Maintainability in Software Product Lines**, measures the Maintainability Index of each feature using an SPL case. Alongwith that, improving feature Maintainability Index and Optimizing Maintainability Index has also been discussed. This work proposed the use of Maintainability Index to take design decisions that would enhance global maintainability.

O. Panchenko et.al [14], in his thesis, **Quality Metrics for Maintainability of standard software**, aims at researching of metric-based quality indicators in order to be able to assess the most important maintainability aspects of the standard software. The used quality model is derived from the Goal Question Metric approach. After literature research, the quality model was expanded by standard metrics and some special newly invented metrics. Some selected metrics were validated to predict maintainability using experiments. He concluded that metrics are a powerful indicators of assessing maintainability. Some metrics need not show results but it is possible to describe the different aspects of maintainability using metric-based indicators.

H. Chung et.al [11] in **Metrics for Maintainability of class inheritance hierarchies**, suggests that there have been many metrics discovered for object oriented paradigm and deficiencies have been found. Among the various measurement of object oriented characteristics, heuristic metrics for class inheritance hierarchies in design and maintenance have been proposed here which further leads to proposing new metrics for understandability and modifiability of class inheritance hierarchy. It is concluded that the deeper the hierarchy of inheritance tree, the better it is for reusability but worse for maintenance and the shallower the hierarchy, the less the abstraction but better understanding and modifying. Whereas **O. Panchenko** [14] in his thesis discusses that, since the properties of metric depends upon the model where the metric is measured in, all metrics are grouped in sets concerning the model they belong to. According to literatures, two major classes of software measures have been found--- intra-modular measures which is based on measures and inter-modular measures, based on entire software systems. Examples like metric LOC counts the number of lines of code excluding whitespaces and comments, which may take up time and is considered to be an important metric because many other metrics correlate with LOC and thus the approximate value of other more complicated metrics could be easily estimated by LOC. Similarly the problem of code cloning can also be measured with a metric since cloning impacts on the maintainability of software quality and increases work. Various techniques for clone findings are token based, string based and parse tree based. Thus choosing of the technique should be made according to the goal of measurement. Metric CLON (CLONICITY), is a ratio of LOC in all detected clones to the total LOC. This metric gives an idea about usage of copy paste in development process and eventually the redundancy of the final product.

D. Coleman et al [15] in **Using Metrics to Evaluate Software System Maintainability**, discusses how automated software maintainability analysis can be used to guide software-related decision making and have applied metrics-based software maintainability models to 11 industrial software systems and used the results for fact-finding and process-selection decisions. **Maintainability** is defined as, ***"The ease with which a software system or component can be modified to correct faults, improve performance or other attributes, or adapt to a changed environment."*** The five methods for quantifying software maintainability are *Hierarchical multidimensional assessment models, Polynomial regression models, Aggregate complexity measure, Principal components analysis and Factor analysis*. An objective of the quality model could be products, processes or projects. Most of the models suggest a decomposition principle, hierarchical quality models organized in a top-down fashion. Nevertheless, bottom-up approach is useful by the metrics validation, when the metrics are already selected. **Goal Question Metric (GQM)** suggests hierarchical top-down approach for selecting appropriate metrics. This model has three levels which are: **Conceptual level (goals), Operational level (questions) and Quantitative level (metrics)**. Optional tool level can also be included here.

D. Batory et al [30] in **Automating Feature Oriented Refactoring of Legacy Applications**, discusses that to reduce costs and risks, developers often take an extractive approach for creating the SPL by refactoring and decomposing one or more legacy applications into features. Refactoring a legacy application into features to create a SPL is a difficult task. It consists of detecting features in the legacy code and their refactoring. Detecting features is an interactive procedure but the refactoring can be automated completely. Authors here proposes a refactoring tool which generates an SPL implemented in Jak or AspectJ based on marked legacy code.

T. Thumm et al [31] in **Variant Preserving Refactoring in Feature Oriented Software Product Lines**, proposes a variant-preserving for SPLs to ensure the validity of all SPL variants after refactoring and discussed the generalizability of this approach for annotative SPLs in general. **M. Kuhlemann** et. al [32] in **Refactoring Feature Modules**, introduces the notion of refactoring feature modules (RFMs) that extend feature modules with refactorings and conclude how RFMs reduce incompatibilities and facilitates reuse based on case studies. An RFM automatically alters the structure of programs which are composed from feature modules. VAMPIRE is a tool used in decomposition and reuse for implementing refactoring. Information

on all SPLs can be found at [33]. FeatureIDE is an open source framework of an Integrated Development Environment for software product line engineering based on Feature-Oriented Software Development (FOSD) as presented in the paper **Feature IDE: An Extensible Framework For Feature Oriented Software Development** by **T. Thum.** et al [34]. FeatureIDE supports the entire life-cycle of a product line in a coherent tool infrastructure, like domain analysis and feature modeling, design, implementation and maintenance. A DeltaJ program is controlled by application condition which is a propositional formula over the available features of a software product line, which decides if a certain delta module participates in the generation of a specific product. Automated program transformations can be used to support the development and evolution of programs, especially DeltaJ using Refactorings, as mentioned by **O. Richers** in his thesis, **Thesis on Transformation and Evolution of DeltaJ Software Product Lines** [25].

K. Raheja et al [35], in their paper, **An Emerging Approach towards Code Clone Detection: Metric Based Approach on Byte Code**, discusses an approach MCD Finder to detect semantic clones to some extent. It is used for calculating metrics for Java program. It uses byte code to calculate the metrics of Java programs, instead of using any transformed representation. The reason for using byte code is that it is platform independent and represents the unified structure of the code which makes its use more efficient. **H.H Mui** et al [36] discusses in **Studying Late Propagations in Code Clone Evolution Using Software Repository Mining**, the approach to extract Late Propagations (one of the clone evolution pattern) from software repositories. The cases have been studied and have been helpful in deriving propagation time, clone dispersion and effects of LPs on the software. In the surveys it has also come under notice that feature modules cannot remove code clone from an implementation. Thus, the design of a SPL always starts from a base feature module which contains common parts of all products. Furthermore, a feature module is intended to represent exactly one product feature.

4. MOTIVATION/ JUSTIFICATION AND RELEVANCE

The basic purpose of evolution and existence of **Software Product Lines** (SPL) are its **user satisfaction, reusability and maintainability**. There are different methodologies for Software Product Lines such as **Feature Oriented Programming** (FOP) and **Delta Oriented Programming** (DOP).

Feature Oriented Programming is where features exist and represented by feature model. A particular software can be propagated further using various features. But the main limitation which was seen here in this approach was that new features can be added but they cannot be removed without disturbing the structure of product line. To overcome this limitation, another approach was developed called Delta Oriented Programming. Here the whole product line is developed using small delta modules and any changes can be added modified or removed just by changing these delta modules which in turn doesn't affect the structure of Software Product Lines.

SPL has been developed to control variability and commonality among the system. Object Oriented paradigm suffers with the problem of Code Clones which adversely affects the software qualities especially Maintainability. Literature survey reveals that this problem persists in FOP too. As the Software Product Line expands, unknowingly lot of Code Clones are introduced in the source code. Code Clones are the codes which are repeated or duplicated in other parts of the source code as the functionality of program increases. Thus Code Clones deteriorates the performance and functionality and introduces errors which adversely affects the Software Product Lines. In the existent literatures, we have found that Feature Oriented Programming was able to remove Object Oriented Code Clones but in effect it introduced Feature Oriented Programming related Code Clones. Till now, no such evidence has been found in the literatures which have justified the occurrence of Code Clones in Delta Oriented Programming. Hence, the focus here is on the Code Clones in Feature Oriented Programming. Thus, the existence of Code Clones in Feature Oriented Programming effects Software Product Lines which in turn affects its various quality factors such as reliability, usability, flexibility, maintainability etc. **Maintainability** is an important factor which we are considering here since the more Code Clones are found in the source code for any Software Product Lines, the more difficult it becomes to maintain Software Product Lines which in turn leads to higher cost. Thus, it becomes crucial to address the code clones as the SPL evolves, since as the SPL grows, its maintainability

also becomes important and cost factor can considerably be reduced if the clones are nipped at the early stages.

This is a key motivation to study the impact of Code Clones on Maintainability of SPL using existing metrics and quality models. Till now, no such study has been conducted which will quantify the effect of code clones on the maintainability factor of SPL and establishes a relation between them. Hence it becomes important to derive the conclusion about the extent to which the Maintainability of SPL is affected due to the presence of code clones. The most suitable remedial measure for the problem of Code Clones is Refactoring.

In our work proposed for research, 10-15 cases of FOP will be taken and studied upon. Till now, no cases are available for DOP , thus our cases are limited to FOP but we shall try to inquire about the DOP code clones and if successful, we shall try to incorporate that in our study too. After studying the limitations of existing models and metrics used in the development process of software product lines, a new model will be designed.

After establishing the relationship between Maintainability and effect of Code Clones, using the proposed model, next step will be to remove those code clones using Refactoring technique (with the help of existing Refactoring methods) which is a known technique for removing code clones.

And lastly, once the code has been refactored, the impact of Maintainability and the refactored code will be assessed.

This study focuses on ascertaining the relationship between the Maintainability as a prime factor for Software quality and Code Cloning in Software Product Lines using the proposed model and also reviewing the impact of Refactored code and Maintainability of Software Product Lines.

5. OBJECTIVES

After the review of relevant research papers, the main objectives under the proposed work are summarized as follows:

1) To make a detailed study of Software Product Line methodology (Feature Oriented Programming and Delta Oriented Programming), Code Cloning problem in SPL, Refactoring technique as a remedial measure for Code Clones, the various aspects of software qualities especially Maintainability, the metrics related to SPL and the Quality Models.

2) To observe and analyze the tools related to Code Cloning, SPL development, Refactoring and statistical tools that might be used.

3) To collect the existing set of Software Product Lines for study, identifying the Code Clones in them using appropriate tool and classify them

4) To identify and analyze the limitations of the existing metrics and models with respect to Maintainability.

5) To design and implement the model with respect to Maintainability.

6) To analyse the impact of Code Clones on the Maintainability using the proposed model and establish the relation between them and asses it.

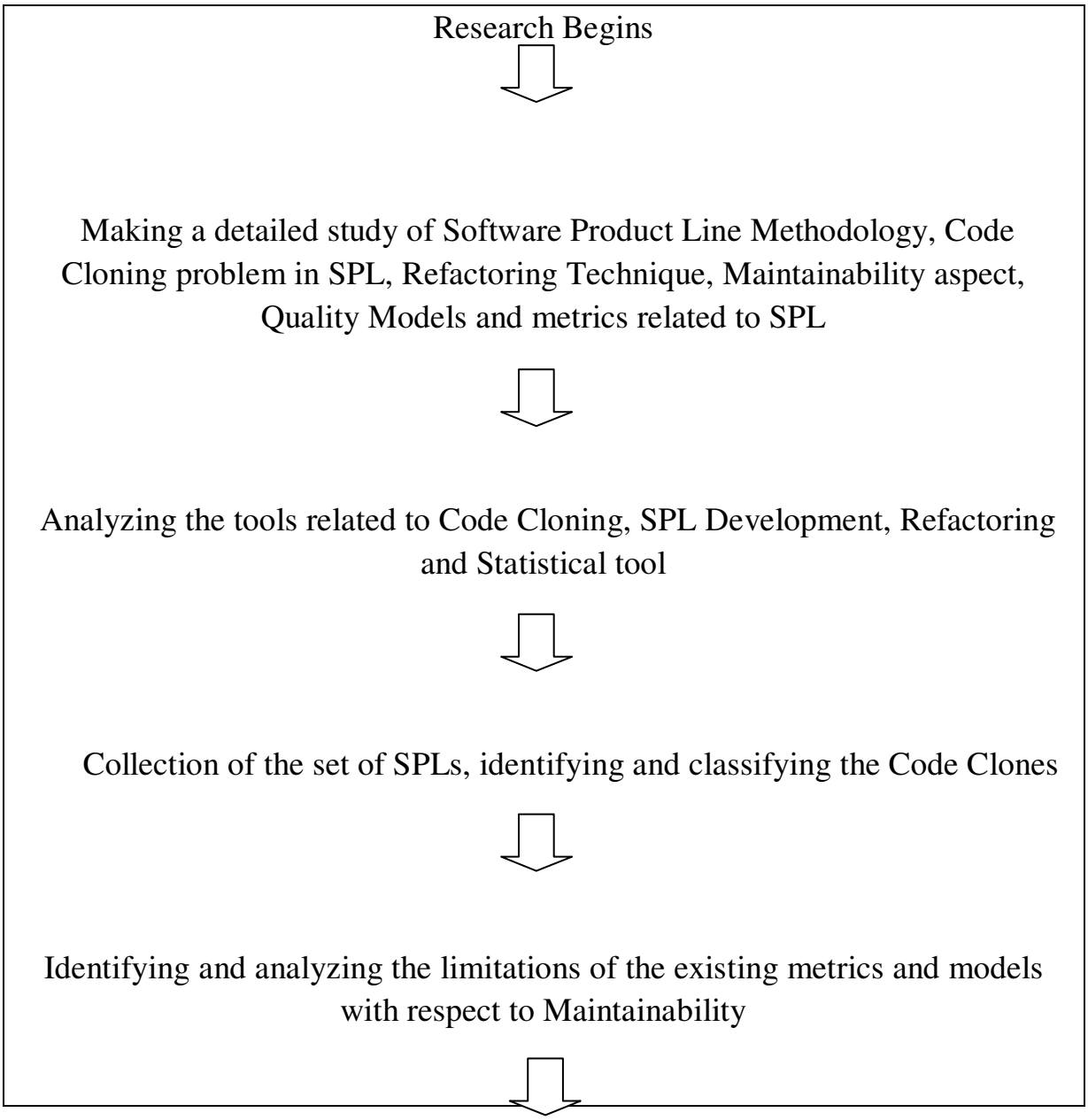
7) To refactor the identified code clones in the collected set of SPLs and review the impact of modified code on Maintainability.

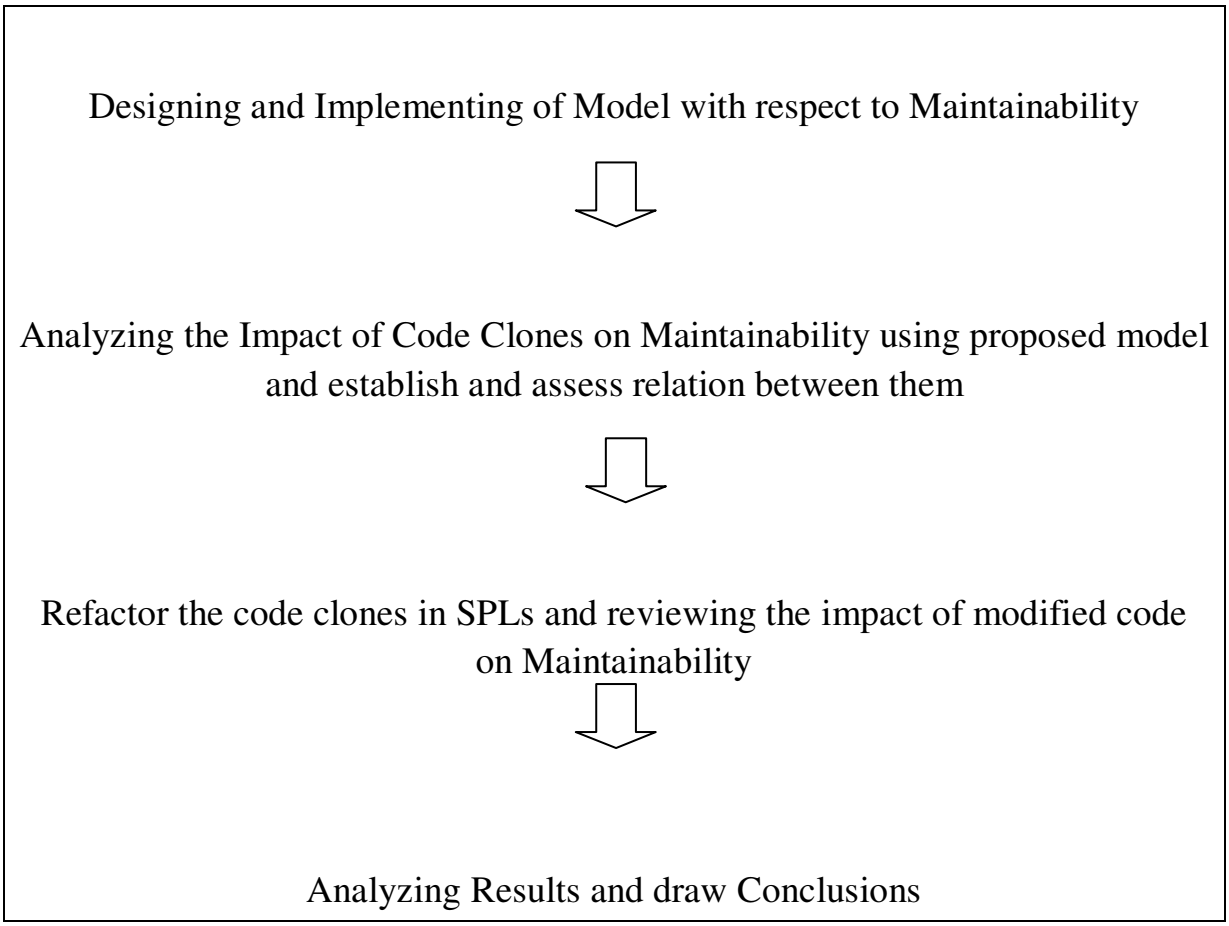
8) To analyze the results and draw conclusions.

6. PLAN OF WORK AND METHODOLOGY

The research process will include following phases for accomplishing the objective of research to study and analyze the data.

6.1 Methodology


Research Begins
Making a detailed study of Software Product Line Methodology, Code Cloning problem in SPL, Refactoring Technique, Maintainability aspect, Quality Models and metrics related to SPL
Analyzing the tools related to Code Cloning, SPL Development, Refactoring and Statistical tool
Collection of the set of SPLs, identifying and classifying the Code Clones
Identifying and analyzing the limitations of the existing metrics and models with respect to Maintainability


Designing and Implementing of Model with respect to Maintainability
Analyzing the Impact of Code Clones on Maintainability using proposed model and establish and assess relation between them
Refactor the code clones in SPLs and reviewing the impact of modified code on Maintainability
Analyzing Results and draw Conclusions

6.2 Workplan

Phase 1 : To study about Software Product Line Methodology, Code Cloning Refactoring, Maintainability aspects, metrics and quality models.

Phase 2 : To analyze tools related to code cloning, SPL development, refactoring and any other statistical tool

Phase 3 : Collect existing SPLs for study, identify code clones in them and classify the clones

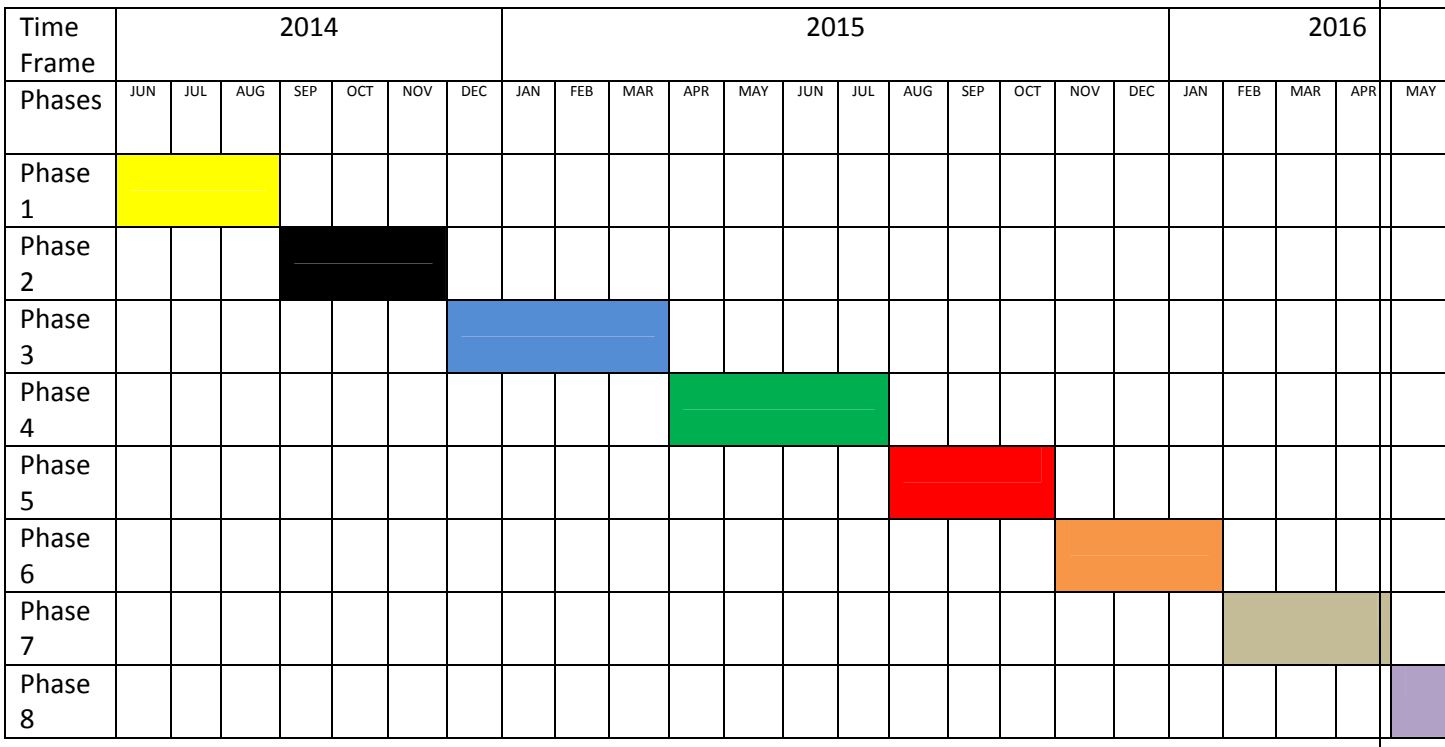
Phase 4 : Identify and analyze limitations of existing models and metrics

Phase 5 : Design a new Model

Phase 6 : Analyse and assess impact of code clones on Maintainability of SPL using the proposed model

Phase 7 : Refactor the code clones in SPLs and review the impact of modified code on Maintainability

Phase 8: Analyze results, draw conclusion and thesis submission



7. EXPECTED OUTCOMES OF THE PROPOSED RESEARCH WORK

a) We wish to design and implement a model and analyze the extent of impact of Code Clones on the Maintainability of Software Product Line.

b) We aspire to observe that to what level the Refactoring technique is helpful in removing Code Clones and in improving the Maintainability in a controlled environment.

8. PLACE OF WORK AND FACILITIES AVAILABLE

The experimental setup for implementation of the tool for code clone detection and identifying the quality model development shall be carried out in the Information Technology Lab in THE IIS UNIVERSITY Campus. Tools for identifying Code Clones would include Code Clone detection tools such as CCFinder Tool, for refactoring such as VAMPIRE tool, and for identifying the appropriate metrics for assessing the maintainability factor quality models such as GQM Model, Variability Model and Network Model etc. can be used.

9. REFERENCE, BIBLIOGRAPHY, WEBLIOGRAPHY

1) S. R. Schach, and A. Tomer. "**Development/maintenance/reuse: Software evolution in product lines**". in: P. Donohoe (Ed.). In *Proceedings of the First Software Product Line Conference : experience and research directiorns*. Kluwer Academic Publishers. pp. 437-450. Norwell, MA, USA, 2000.

2) Mnkandla, E." **About Software engineering Frameworks and Methodologies**". In *Proceeding of IEEE AFRICON 2009*. pp 279-299. Nairobi, Kenya. 23-25 Sep., 2009.
3) A. Aitken ,V. Lingo. "**Comparative Analysis of Traditional Software Engineering and Agile Software Development**". *46th Hawaii International Conference on System Sciences*. pp 4751-4760, 2013.
4) M. Eriksson. "**An Introduction to Software Product Line Development**". *Proceedings of Umeå's Seventh Student Conference in Computing Science UMINF*. Volume 3, Issue 5. pp. 26-37, 2003.
5) R. Mitschke and M. Eichberg. "**Supporting the evolution of software product lines**". In *4th ECMDA-Traceability Workshop (ECMDA-TW)* . pp 87-96, 2008.
6) S. Apel and C. Kastner. "**An overview of feature-oriented software development**". In *Journal of Object Technology*. Volume 8, Issue 5. pp. 49-84, 2009.
7) S. Schulze, O. Richers, and I. Schaefer. "**Refactoring delta-oriented software product lines**". In *AOSD*. pp. 73-84. ACM, 2013.
8) I. Schaefer, L. Bettini, F. Damiani, and N. Tanzarella. "**Delta-oriented programming of software product lines**". In *SPLC*. pp. 77-91. Springer, 2010.
9) S. Schulze, S. Apel and C. Kästner. "**Code Clones in Feature-Oriented Software Product Lines**". In *Proceedings of the 9th International Conference on Generative Programming and Component Engineering (GPCE)* (Eindhoven, The Netherlands), New York, NY, USA. pp. 103-112. ACM Press, October 2010.
10) S. Schulze."**Analysis and Removal of Code Clones in Software Product Lines**". Dissertation. University of Magdeburg, 2012.
11) I. Heitlager, T. Kuipers, and J. Visser. "**A practical model for measuring maintainability**" In *6th Int. Conference on the Quality of Information and Communications Technology (QUATIC 2007).* pp. 30-39. IEEE Computer Society, 2007.
12) F. T. Sheldon, K. Jerath, and H. Chung. "**Metrices for maintainability of class inheritance hierarchies**". In *Journal of software maintenance and evaluation: Research and practice*. Volume 14, Issue 3. pp 147-160, 2002.

13) G. Aldekoa, S. Trujillo, G. Sagardui, and O. Dıaz. "**Experience Measuring Maintainability in Software Product Lines**". In *Proceedings of the XI Jornadas de Ingenier´ıadel Software Bases de Datos (JISBD 2006)*. pp. 173-182, 2006.

14) T. Yi, "**Comparison Research of Two Typical UML-Class-Diagram Metrics: Experimental Software Engineering**". In *International Conference on Computer Application and System Modeling*. Volume 12. pp. 86-90. Taiyuan, 2012.

15) O. Panchenko. "**Quality Metrics for Maintainability of Standard Software**". Master Thesis. University of Potsdam, 2006.

16) D. M. Coleman, D. Ash, B. Lowther and P. W. Oman, "**Using metrics to evaluate software system maintainability**". Volume 27, Issue 8. pp. 44–49. IEEE Computer, 1994.

17) V.R. Basili, "**Software modeling and measurement: The Goal Question Metric paradigm**". In *Computer Science Technical Report Series NR: UMIACS-TR*. pp 92-96, 1992.

18) M. Fowler. "**Refactoring: Improving the Design of Existing Code**". Addison-Wesley, 1st Edition. USA, 2000.

19) D. Simon and T. Eisenbarth. "**Evolutionary Introduction of Software Product Lines**". In *Proc. Int'l Software Product Line Conference (SPLC)*. pp 272–282, 2002.

20) M. Svahnberg, J.Bosch. "**Evolution in Software Product Lines**". In *Journal of Software Maintenance: Research and Practice*. Volume 11, Issue 6. pp 391–422. November/December 1999.

21) L. Chen, M. A. Babar and N. Ali. "**Variability Management in Software Product Lines: A Systematic Review**". In *Proceedings of the 13th International Software Product Line Conference.* pp 81-90. Pittsburgh, Pa, 2009.

22) T. Maben and H. Lichter. "**Deficiencies in Feature Models**". In *Workshop on Software Variability Management for Product Derivation-towards Tool Support*. August 2004.

23) C. Gacek and M. Anastasopoulos. "**Implementing Product Line Variabilities**". In *SIG-SOFT Softw. Eng. Notes*. Volume 26, Issue 3. pp 109-117, 2001.

24) T Thümm, C. Kästner, S. Erdweg and N. Siegmund. "**Abstract Features in Feature Modeling**". In *Proceedings of the 15th International Software Product Line Conference (SPLC)* (Munich). pp 191-200. IEEE Computer Society. Los Alamitos, CA, USA, August 2011.

25) A. Haber, H. Rendel, B. Rumpe, and I. Schaefer. "**Delta Modeling for Software Architectures**". In *Tagungsband des Dagstuhl- Workshop MBEES: Modellbasierte Entwicklung eingebetteterSysteme VII*. pp 1-10. Fotiss GmbH, Munich, Germany, February 2011.

26) O. Richers. " **Transformation and Evolution of DeltaJ Software Product Lines**". University of Braunschweig, 2012.

27) F. Takeyama and S. Chiba. "**Feature-Oriented Programming with Family Polymorphism**". In *Proceedings of the International Workshop on Variability & Composition (VariComp)*. pp 1-6. ACM Press, 2012.

28) S. Thummalapenta, L. Cerulo, L. Aversano and M. D. Penta. "**An Empirical Study on the Maintenance of Source Code Clones**". In *Empirical Software Engineering Journal*. Volume 15. pp 1-34, 2010.

29) S. S. Kolesnikov, S. Apel, N. Siegmund, S. Sobernig, C. Kästner, S. Senkaya, "**Predicting Quality Attributes of Software Product Lines Using Software and Network Measures and Sampling".** In *VaMoS '13 The Seventh International Workshop on Variability Modelling of Software-intensive Systems.* ACM. Pisa , Italy. January 23 - 25, 2013.

30) C. Kästner, M. Kuhlemann, and D. Batory. "**Automating Feature-Oriented Refactoring of Legacy Applications**". In *Poster presented at Europ. Conf. Object-Oriented Programming.* 2007.

31) S. Schulze, T. Thum, M. Kuhlemann, and G. Saake. "**Variant-preserving refactoring in feature-oriented software product lines**". In *VaMoS*. pp 73-81. ACM, 2012.

32) M. Kuhlemann, D. Batory, and S. Apel. "**Refactoring feature modules**". In *ICSR*. pp 106-115. Springer, 2009.

33) **http://www.fosd.de/fh**. Last accessed 10.05.2014

34) T. Thüm, C. Kästner, F. Benduhn, J. Meinicke, G. Saake, and T. Leich. "**FeatureIDE: An Extensible Framework for Feature-Oriented Software Development**". In *Science of Computer Programming*, 2012.

35) K. Raheja and R. Tekchandani, "**An Emerging Approach towards Code Clone Detection:Metrics based approach on Byte Code**." In *IJARCSSE.* Volume 3, Issue 5. May, 2013.

36) H. H. Mui, A. Zaidman and M. Pinzger. "**Studying Late Propagations in Code Clone Evolution Using Software Repository Mining**". In *Proceedings of the Eighth International Workshop on Software Clones (IWSC 2014) in conjunction with CSMR-WCRE'14*. February 3rd, Antwerp, Belgium, 2014.